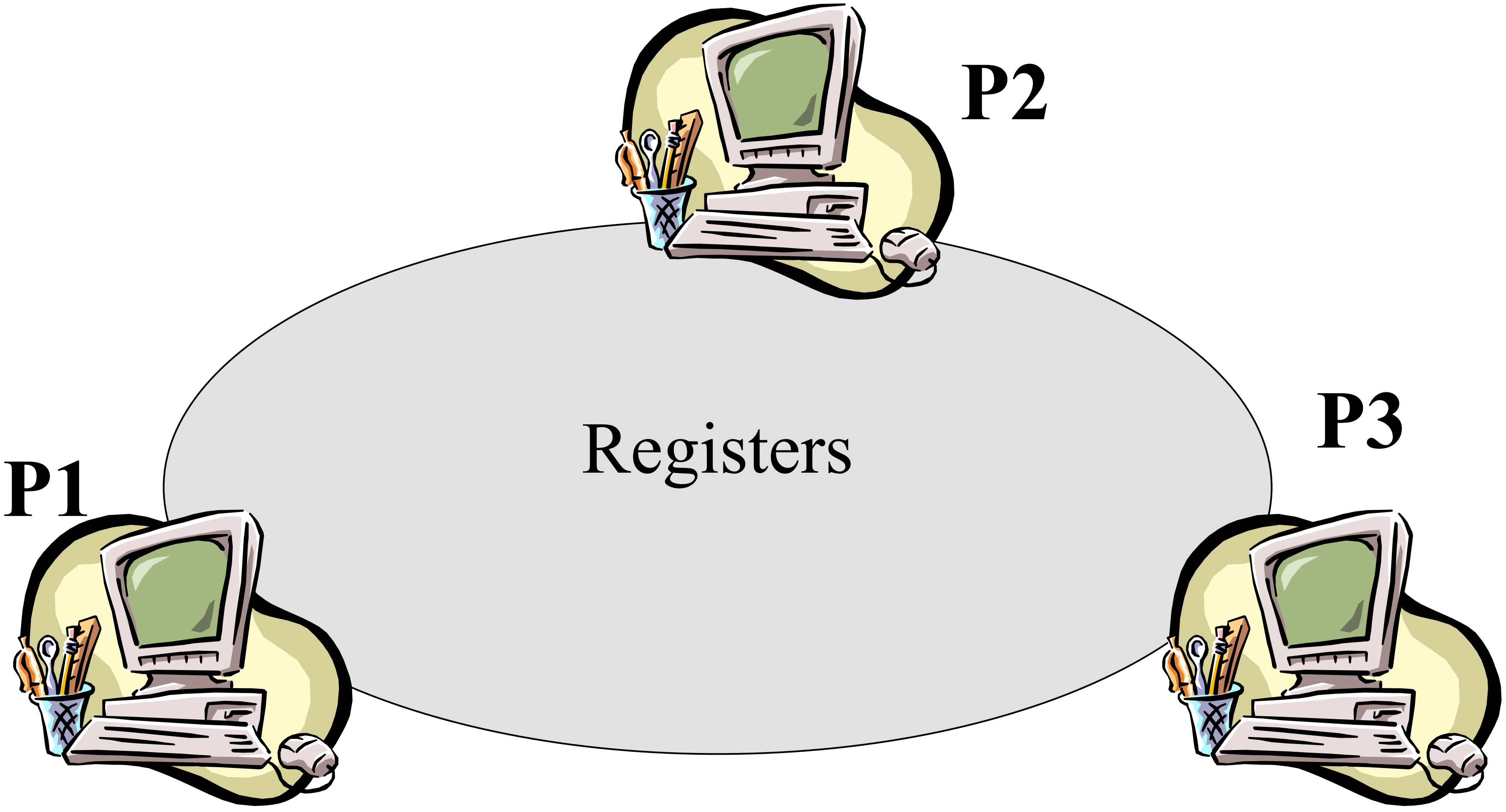
- Shared Memory -

R. Guerraoui
Distributed Programming Laboratory
lpdwww.epfl.ch

The application model



Register (assumptions)

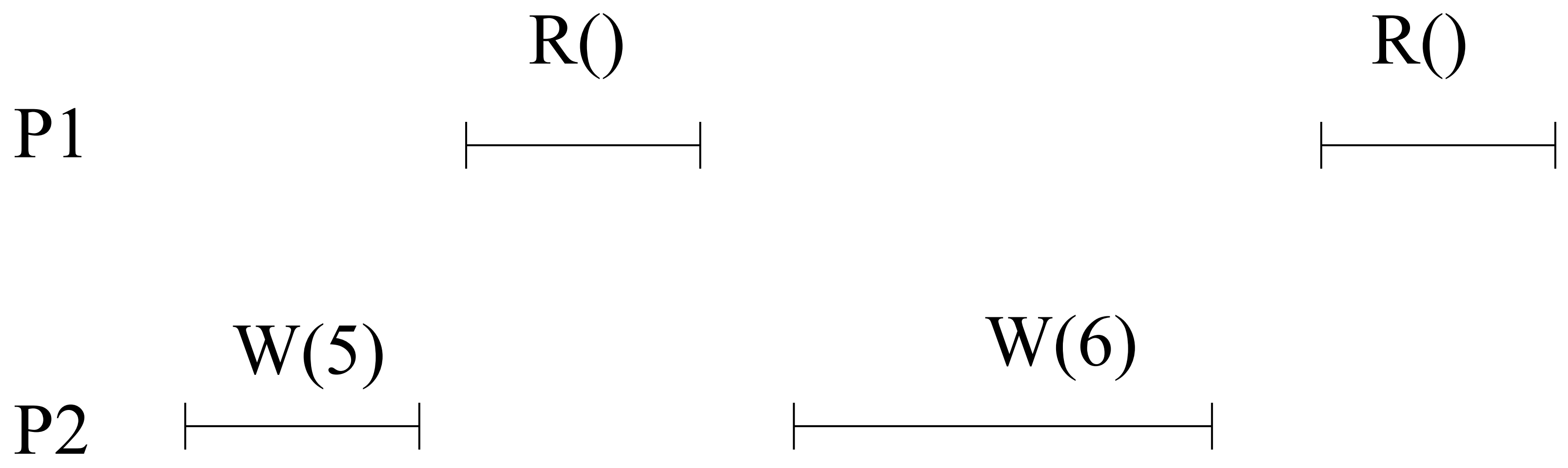
- For presentation simplicity, we assume registers of ***integers***
- We also assume that the initial value of a register is 0 and this value is initialized (written()) by some process before the register is used
- We assume that every value written is ***uniquely*** identified (this can be ensured by associating a process id and a timestamp with the value)

Register: specification

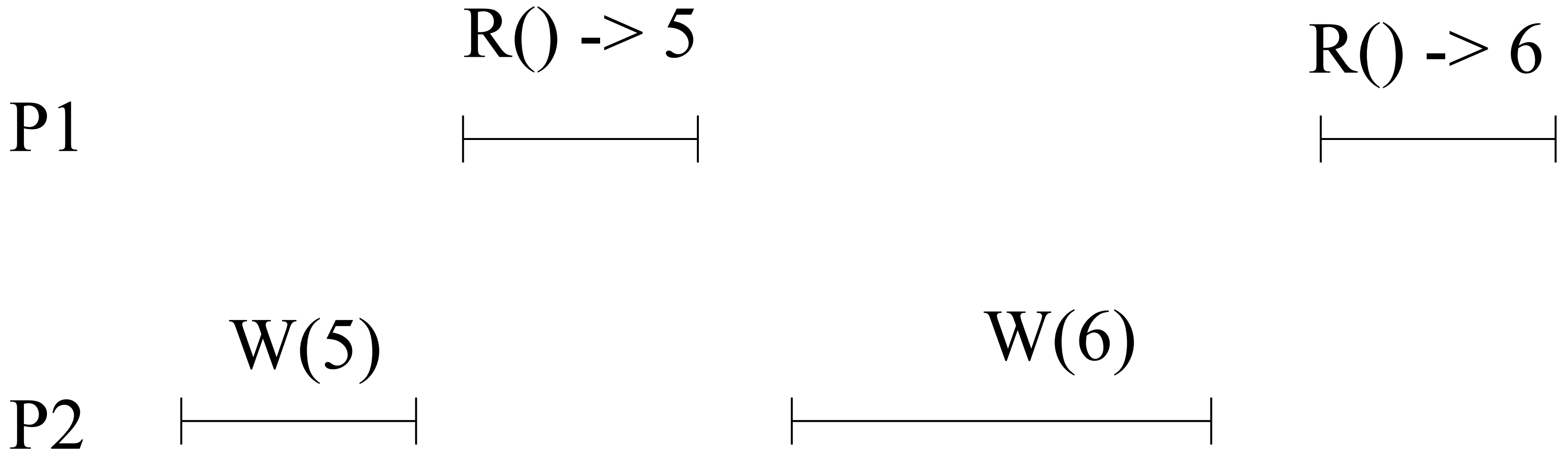
- Assume a register that is local to a process, i.e., accessed only by one process:

- In this case, the value returned by a ***Read()*** is the last value written

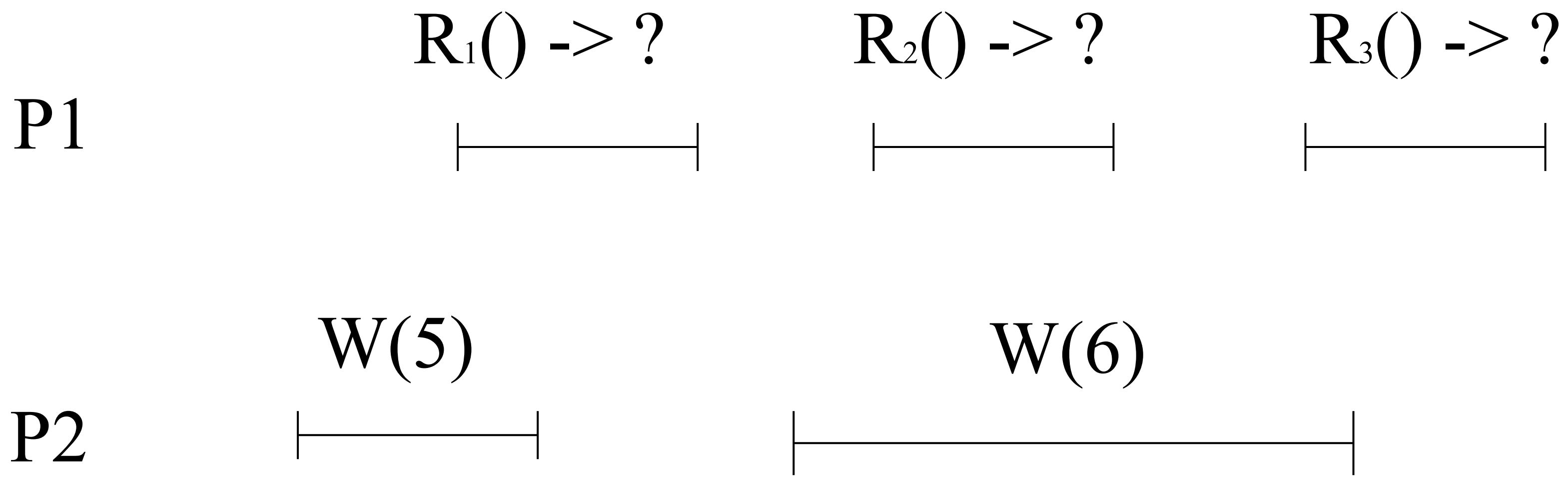
Sequential execution



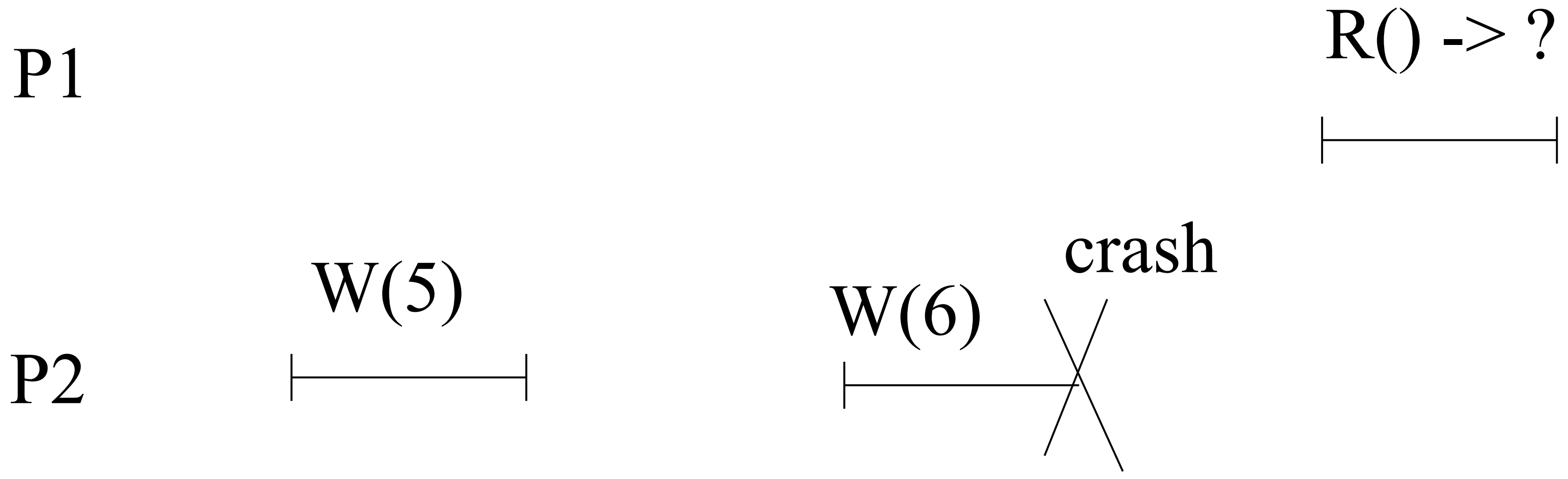
Sequential execution



Concurrent execution



Execution with failures



Regular register

- It assumes only ***one*** writer;
- It provides ***strong*** guarantees when there is no concurrent or failed operations (invoked by processes that fail in the middle)
- When some operations are concurrent, or some operation fails, the register provides ***minimal*** guarantees

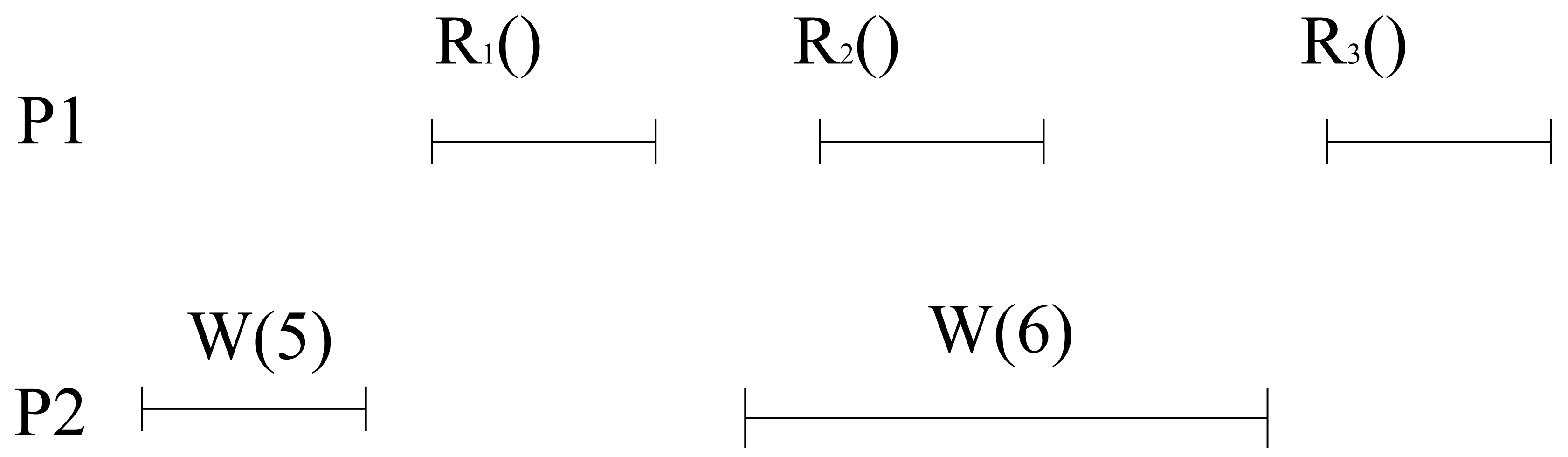
Regular register

- ***Read()*** returns:

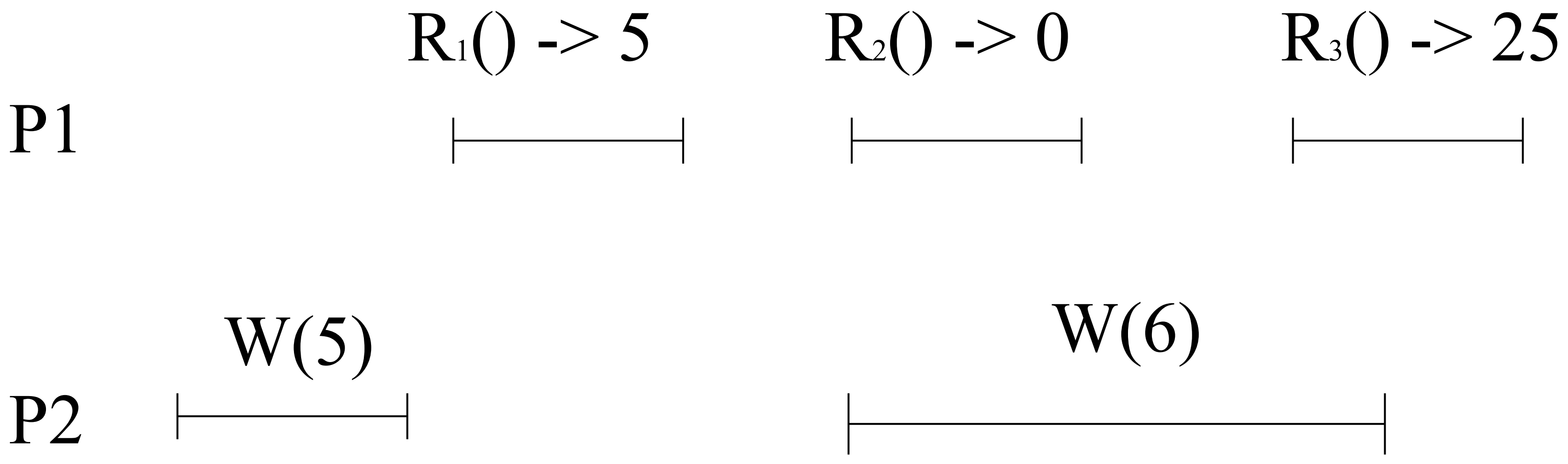
  ✓***the last value*** written if there is no concurrent or failed operations

  ✓and otherwise the last value written or ***any*** value concurrently written, i.e., the input parameter of some ***Write()***

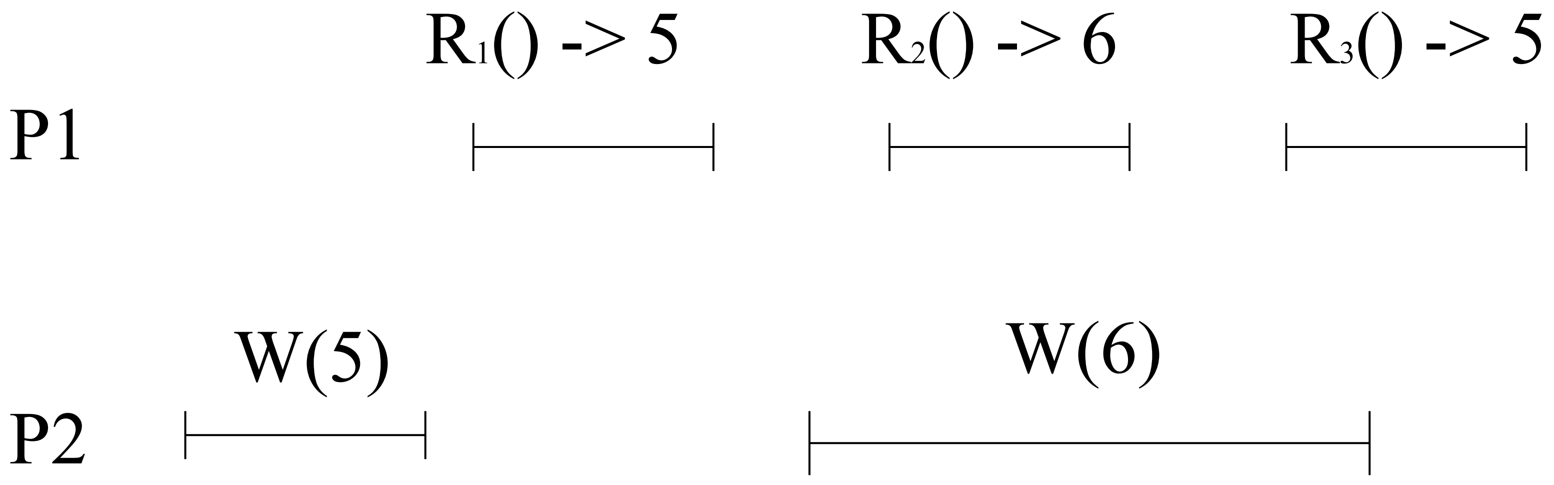
Execution



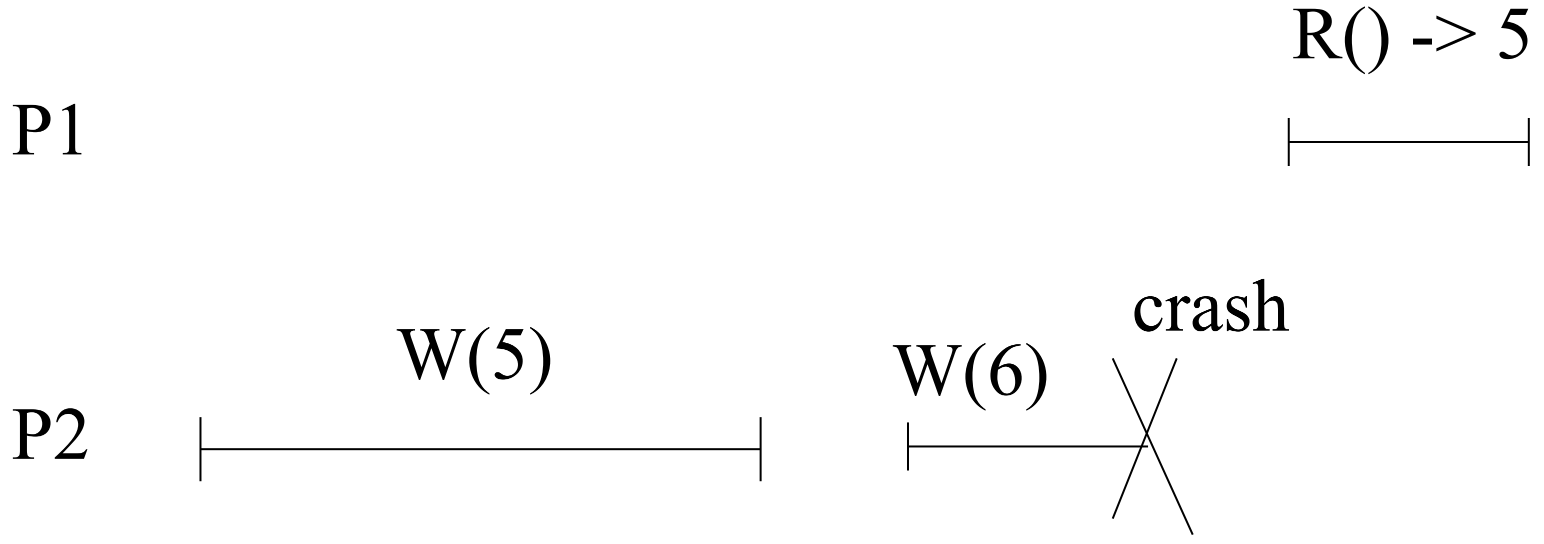
Results 1



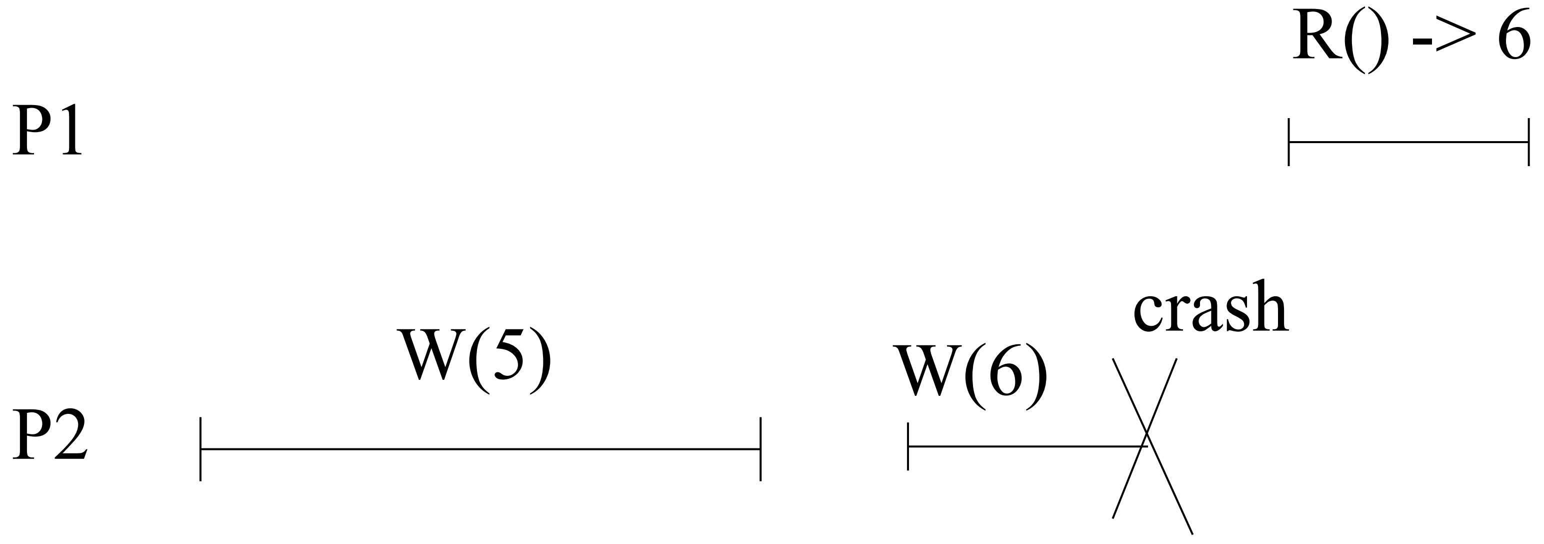
Results 2



Results 3



Results 4



Correctness

- Results 1: non-regular register (safe)
- Results 2; 3; 4: regular register

Regular register algorithms

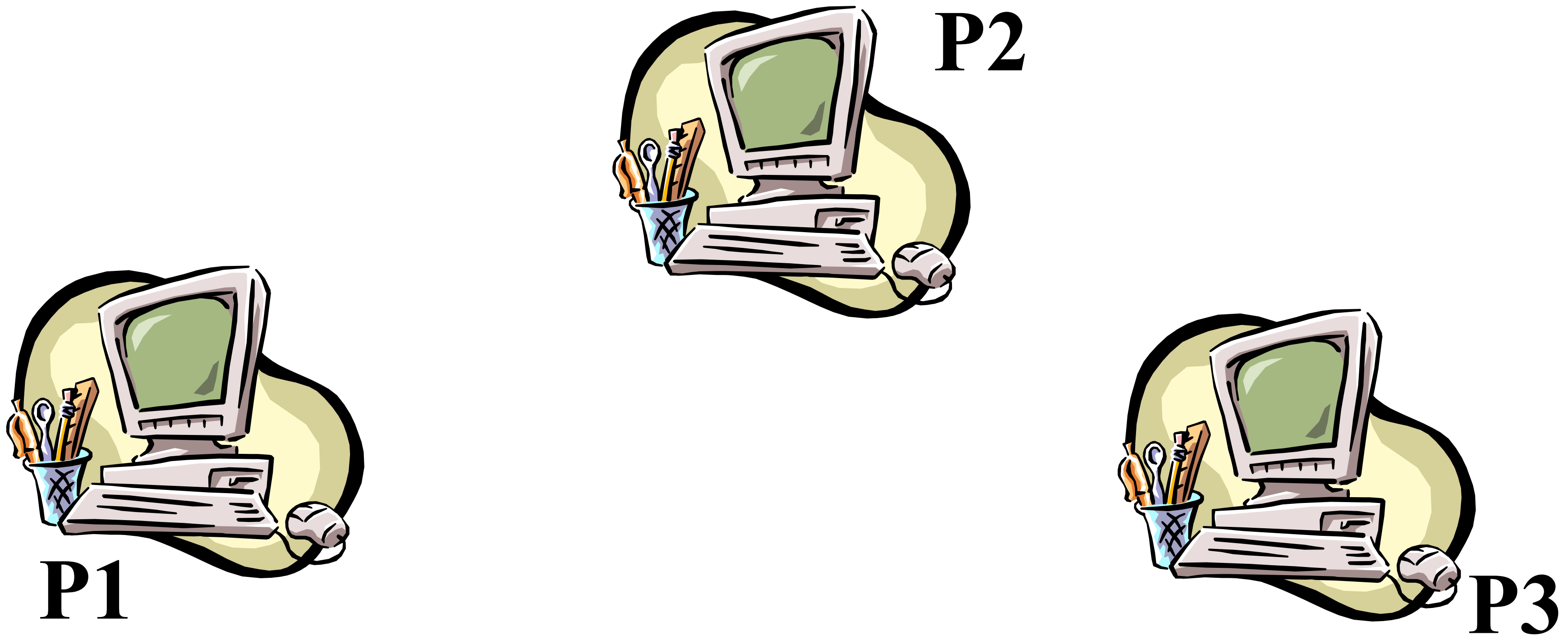
R. Guerraoui

Distributed Programming Laboratory
lpdwww.epfl.ch

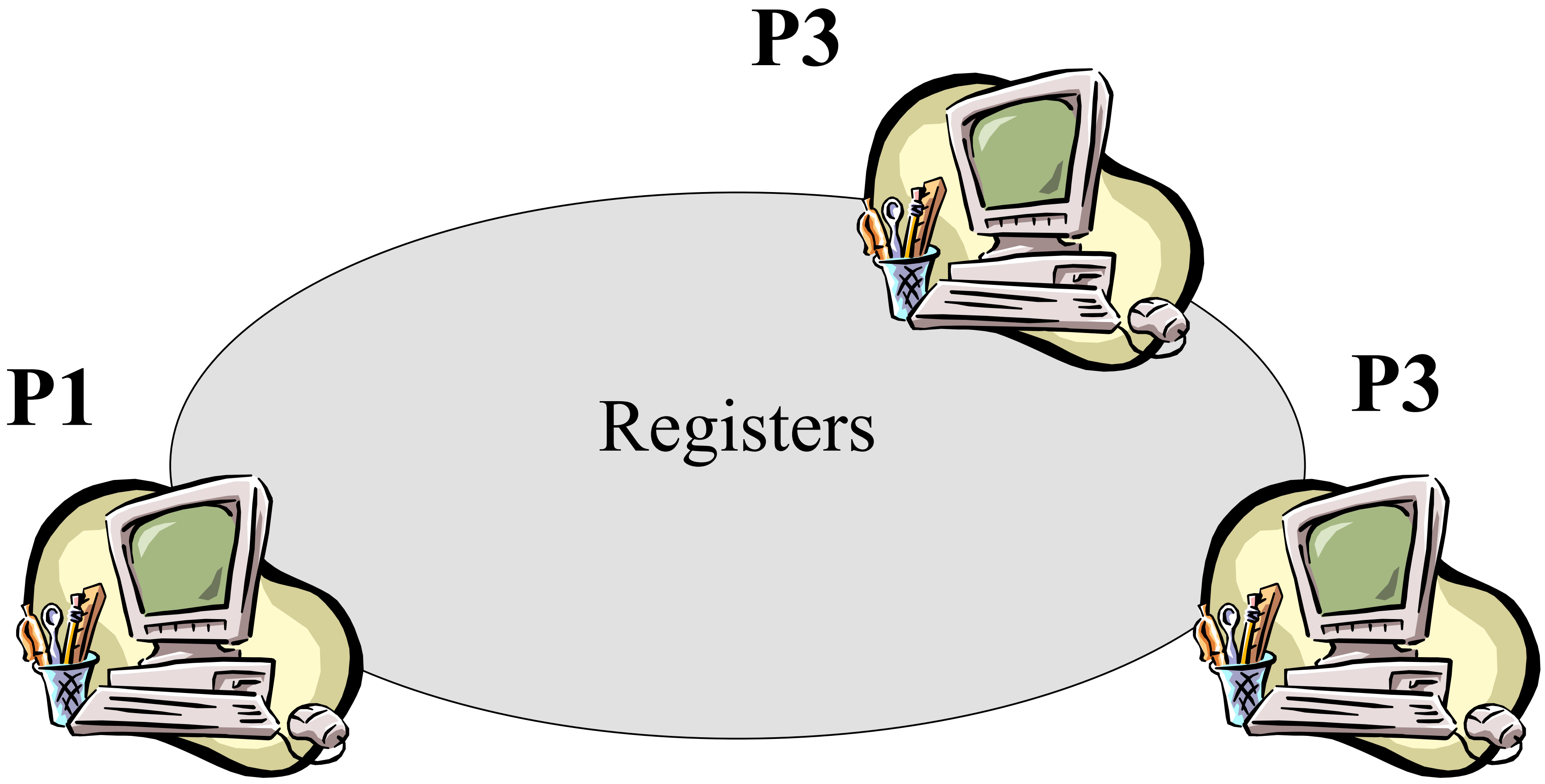
Overview of this lecture

- (1) ***Overview of a register algorithm***
- (2) ***A bogus algorithm***
- (3) ***A simplistic algorithm***
- (4) ***A simple fail-stop algorithm***
- (5) ***A tight asynchronous lower bound***
- (6) ***A fail-silent algorithm***

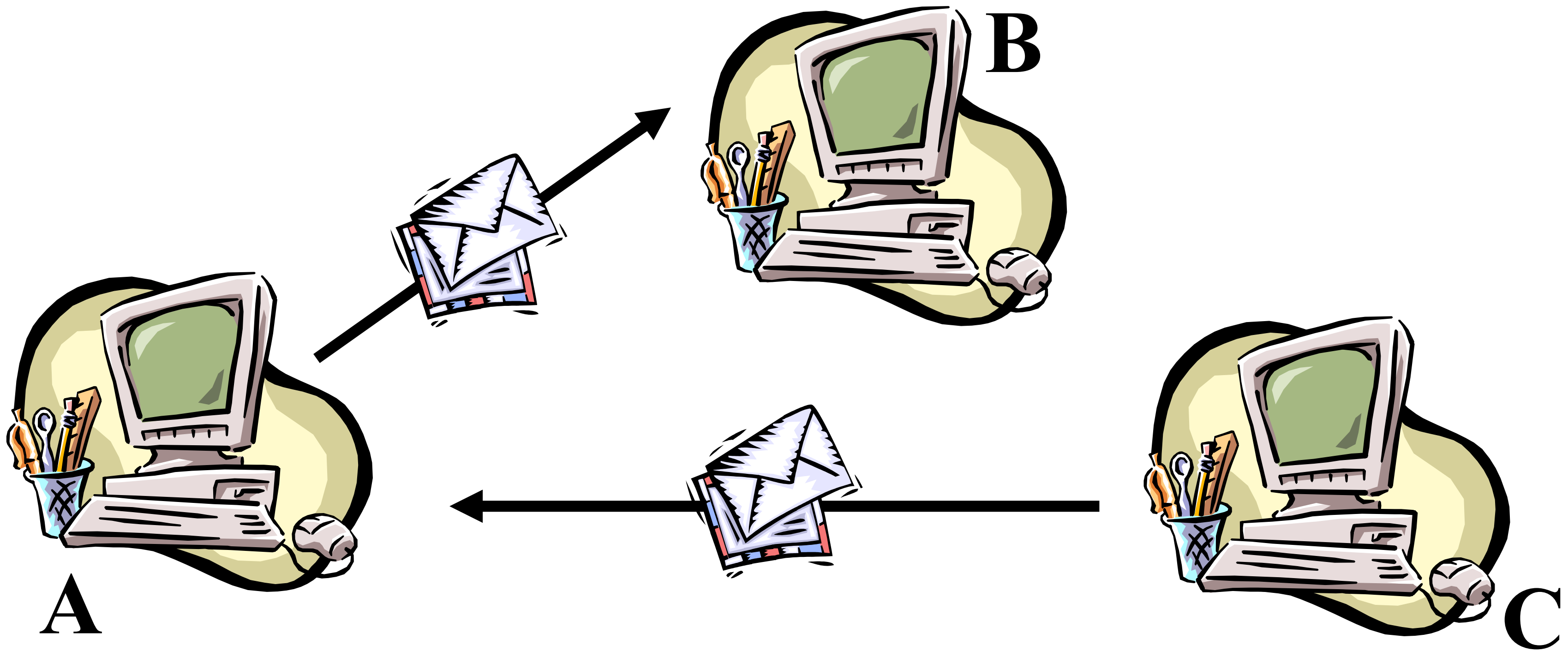
A distributed system



Shared memory model



Message passing model



Implementing a register

- From message passing to shared memory
- Implementing the register comes down to implementing ***Read()*** and ***Write()*** operations at every process

Implementing a register

- Before returning a ***Read()*** value, the process must communicate with other processes

- Before performing a ***Write(),*** i.e., returning the corresponding ok, the process must communicate with other processes

Overview of this lecture

- (1) ***Overview of a register algorithm***
- (2) ***A bogus algorithm***
- (3) ***A simplistic algorithm***
- (4) ***A simple fail-stop algorithm***
- (5) ***A tight asynchronous lower bound***
- (6) ***A fail-silent algorithm***

A bogus algorithm

- We assume that channels are reliable (perfect point to point links)

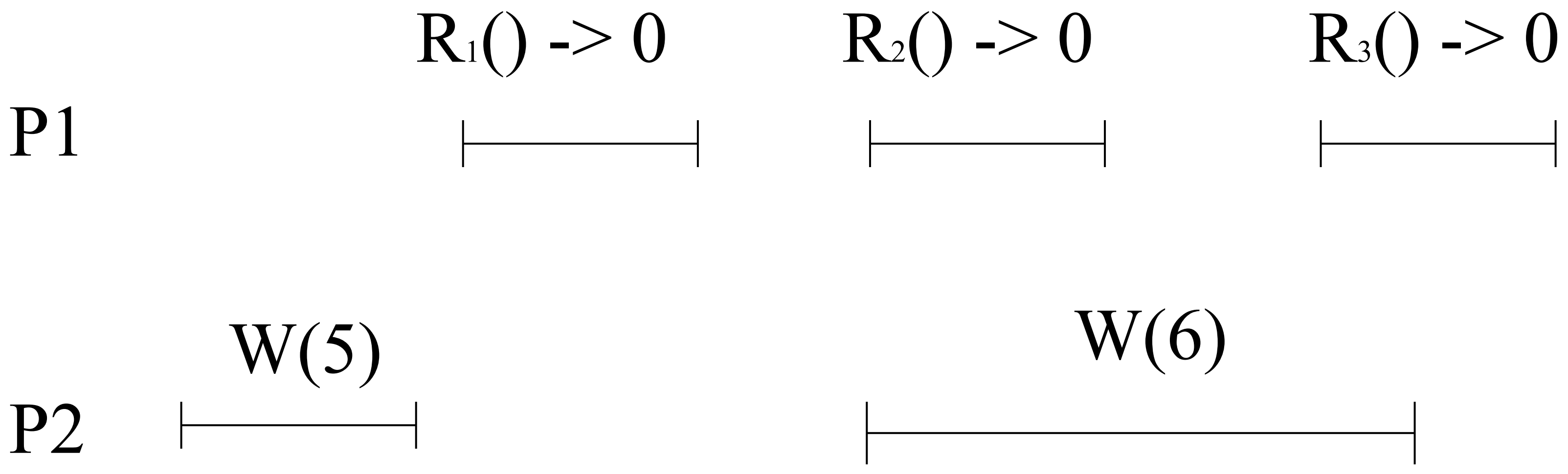
- Every process pi holds a copy of the register value vi

A bogus algorithm

- Read() at pi
  - ✓Return vi
- Write(v) at pi
  - ✓ vi := v
  - ✓Return ok

- The resulting register is live but not safe:
  - ✓Even in a sequential and failure-free execution, a ***Read()*** by pj might not return the last written value, say by pi

No safety



Overview of this lecture

- (1) ***Overview of a register algorithm***
- (2) ***A bogus algorithm***
- (3) ***A simplistic algorithm***
- (4) ***A simple fail-stop algorithm***
- (5) ***A tight asynchronous lower bound***
- (6) ***A fail-silent algorithm***

A simplistic algorithm

- We still assume that channels are reliable but now we also assume that no process fails

- Basic idea: one process, say p1, holds the value of the register

A simplistic algorithm

- Read() at pi
  - ✓ send [R] to p1
  - ✓ when receive [v]
  - ✓ Return v
- Write(v) at pi
  - ✓ send [W,v] to p1
  - ✓ when receive [ok]
  - ✓ Return ok

- At p1:

```
T1:
 when receive [R] from pi
   send [v1] to pi
T2:
  when receive [W,v] from pi
   v1 := v
   send [ok] to pi
```

Correctness (liveness)

- By the assumption that
  - ✓ (a) no process fails,
  - ✓ (b) channels are reliable

no wait statement blocks forever, and hence every invocation eventually terminates

Correctness (safety)

- (a) If there is no concurrent or failed operation, a ***Read()*** returns the last value written
- (b) A ***Read()*** must return some value concurrently written or the last value written
- NB. If a ***Read()*** returns a value written by a given ***Write(),*** and another ***Read()*** that starts later returns a value written by a different ***Write(),*** then the second ***Write()*** cannot start after the first ***Write()*** terminates

Correctness (safety – 1)

- (a) If there is no concurrent or failed operation, a ***Read()*** returns the last value written
  - Assume a Write(x) terminates and no other Write() is invoked. The value of the register is hence x at p1. Any subsequent Read() invocation by some process pj returns the value of p1, i.e., x, which is the last written value

Correctness (safety – 2)

- (b) A ***Read()*** returns the previous value written or the value concurrently written
  - Let x be the value returned by a Read(): by the properties of the channels, x is the value of the register at p1. This value does necessarily come from a concurrent or from the last Write().

What if?

Processes might crash?

If p1 crashes, then the register is not live (wait-free)

If p1 is always up, then the register is regular and wait-free

Overview of this lecture

- (1) ***Overview of a register algorithm***
- (2) ***A bogus algorithm***
- (3) ***A simplistic algorithm***
- (4) ***A simple fail-stop algorithm***
- (5) ***A tight asynchronous lower bound***
- (6) ***A fail-silent algorithm***

A fail-stop algorithm

- We assume a ***fail-stop*** model; more precisely:
  - any number of processes can fail by crashing (no recovery)
  - channels are reliable
  - failure detection is perfect (we have a perfect failure detector)

A fail-stop algorithm

- We implement a ***regular*** register
  - every process pi has a local copy of the register value vi
  - every process reads ***locally***
  - the writer writes ***globally,*** i.e., at all (non-crashed) processes

A fail-stop algorithm

- Write(v) at pi
  - send [W,v] to all
  - for every pj, wait until either:
    - receive [ack] or
    - suspect [pj]
  - Return ok

- At pi:

  when receive [W,v] from pj

  vi := v

  send [ack] to pj

- Read() at pi
  - Return vi

Correctness (liveness)

- ✓ A Read() is local and eventually returns
- ✓ A Write() eventually returns, by the
  - (a) the strong completeness property of the failure detector, and
  - (b) the reliability of the channels

Correctness (safety – 1)

- (a) In the absence of concurrent or failed operation, a Read() returns the last value written
  - Assume a Write(x) terminates and no other Write() is invoked. By the accuracy property of the failure detector, the value of the register at all processes that did not crash is x. Any subsequent Read() invocation by some process pj returns the value of pj, i.e., x, which is the last written value

Correctness (safety – 2)

- (b) A Read() returns the value concurrently written or the last value written
  - Let x be the value returned by a Read(): by the properties of the channels, x is the value of the register at some process. This value does necessarily come from the last or a concurrent Write().

What if?

- Failure detection is not perfect
- Can we devise an algorithm that implements a regular register and tolerates an arbitrary number of crash failures?

Overview of this lecture

- (1) ***Overview of a register algorithm***
- (2) ***A bogus algorithm***
- (3) ***A simplistic algorithm***
- (4) ***A simple fail-stop algorithm***
- (5) ***A tight asynchronous lower bound***
- (6) ***A fail-silent algorithm***

Lower bound

- ***Proposition:*** any wait-free asynchronous implementation of a regular register requires a majority of correct processes

- Proof (sketch): assume a Write(v) is performed and n/2 processes crash, then a Read() is performed and the other n/2 processes are up: the Read() cannot see the value v

- The impossibility holds even with a 1-1 register (one writer and one reader)

The majority algorithm [ABD95]

- We assume that p1 is the writer and any process can be reader
- We assume that a majority of the processes is correct (the rest can fail by crashing – no recovery)
- We assume that channels are reliable
- Every process pi maintains a local copy of the register vi, as well as a sequence number sni and a read timestamp rsi
- Process p1 maintains in addition a timestamp ts1

Algorithm - Write()

- Write(v) at p1
  - ✓ ts1++
  - ✓ send [W,ts1,v] to all
  - ✓ when receive [W,ts1,ack] from majority
  - ✓ Return ok

- At pi
  - ✓ when receive [W,ts1, v] from p1
  - ✓ If ts1 > sni then
    - vi := v
    - sni := ts1
    - send [W,ts1,ack] to p1

Algorithm - Read()

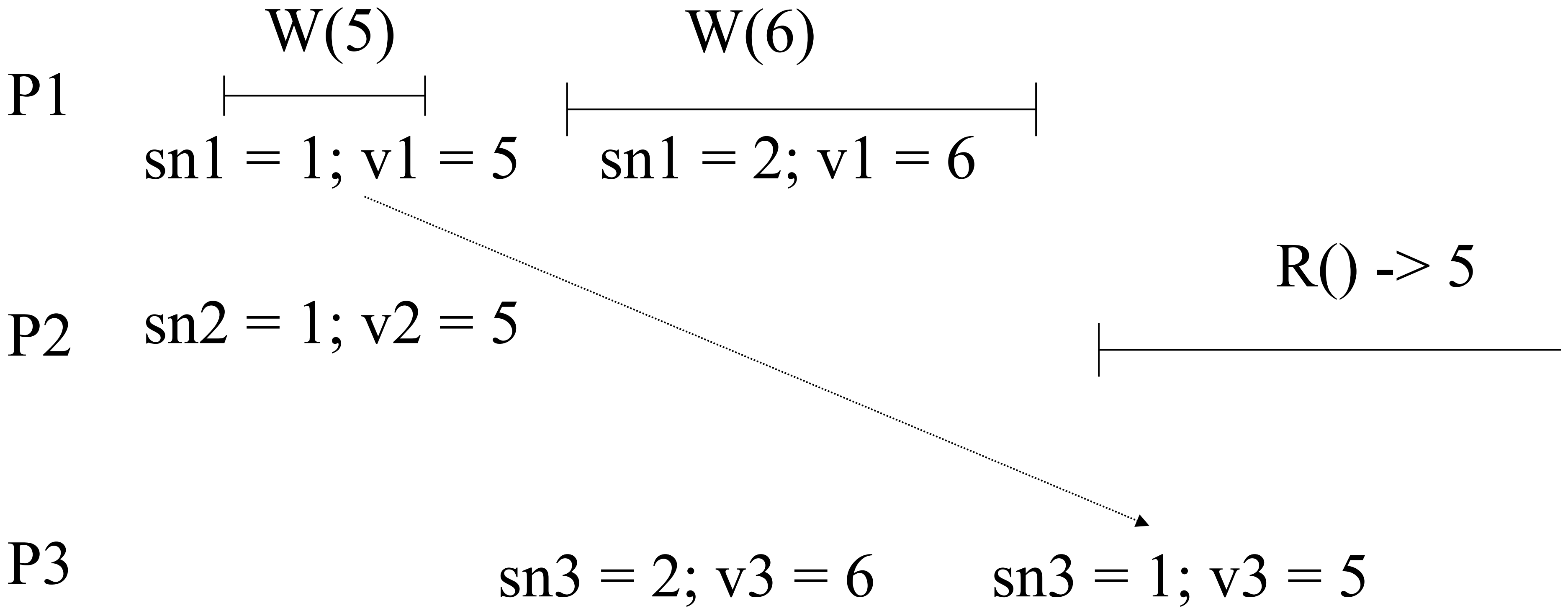
- Read() at pi
  - ✓ rsi++
  - ✓ send [R,rsi] to all
  - ✓ when receive [R, rsi,snj,vj] from majority
  - ✓ v := vj with the largest snj
  - ✓ Return v

- At pi
  - ✓ when receive [R,rsj] from pj
  - ✓ send [R,rsj,sni,vi] to pj

What if?

- Any process that receives a write message (with a timestamp and a value) updates its value and sequence number, i.e., without checking if it actually has an older sequence number

Old writes



Correctness 1

- ✓ Liveness: Any Read() or Write() eventually returns by the assumption of a majority of correct processes (if a process has a newer timestamp and does not send [W,ts1,ack], then the older Write() has already returned)

- ✓ Safety 2: By the properties of the channels, any value read is the last value written or the value concurrently written

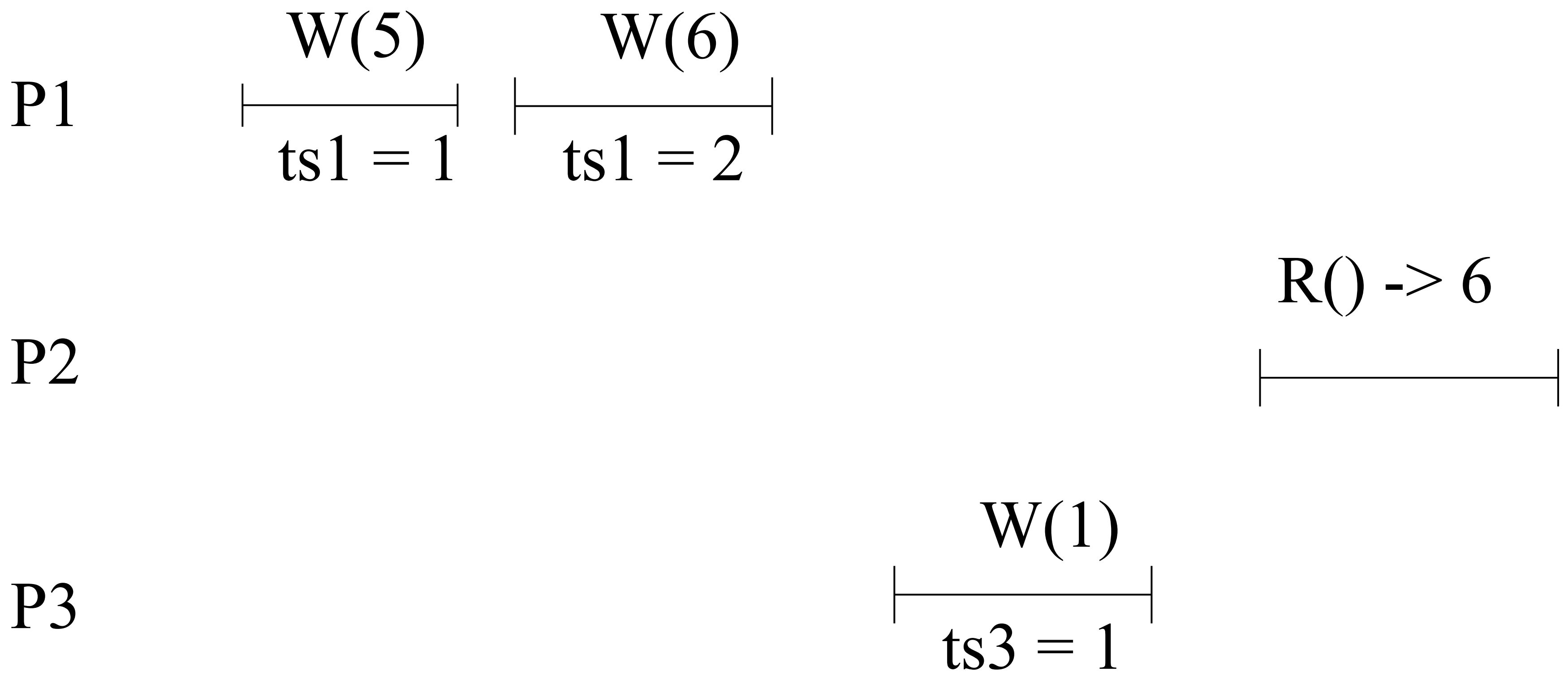
Correctness 2 (safety – 1)

- (a) In the absence of concurrent or failed operation, a Read() returns the last value written
  - Assume a Write(x) terminates and no other Write() is invoked. A majority of the processes have x in their local value, and this is associated with the highest timestamp in the system. Any subsequent Read() invocation by some process pj returns x, which is the last written value

What if?

- Multiple processes can write concurrently?

Concurrent writes



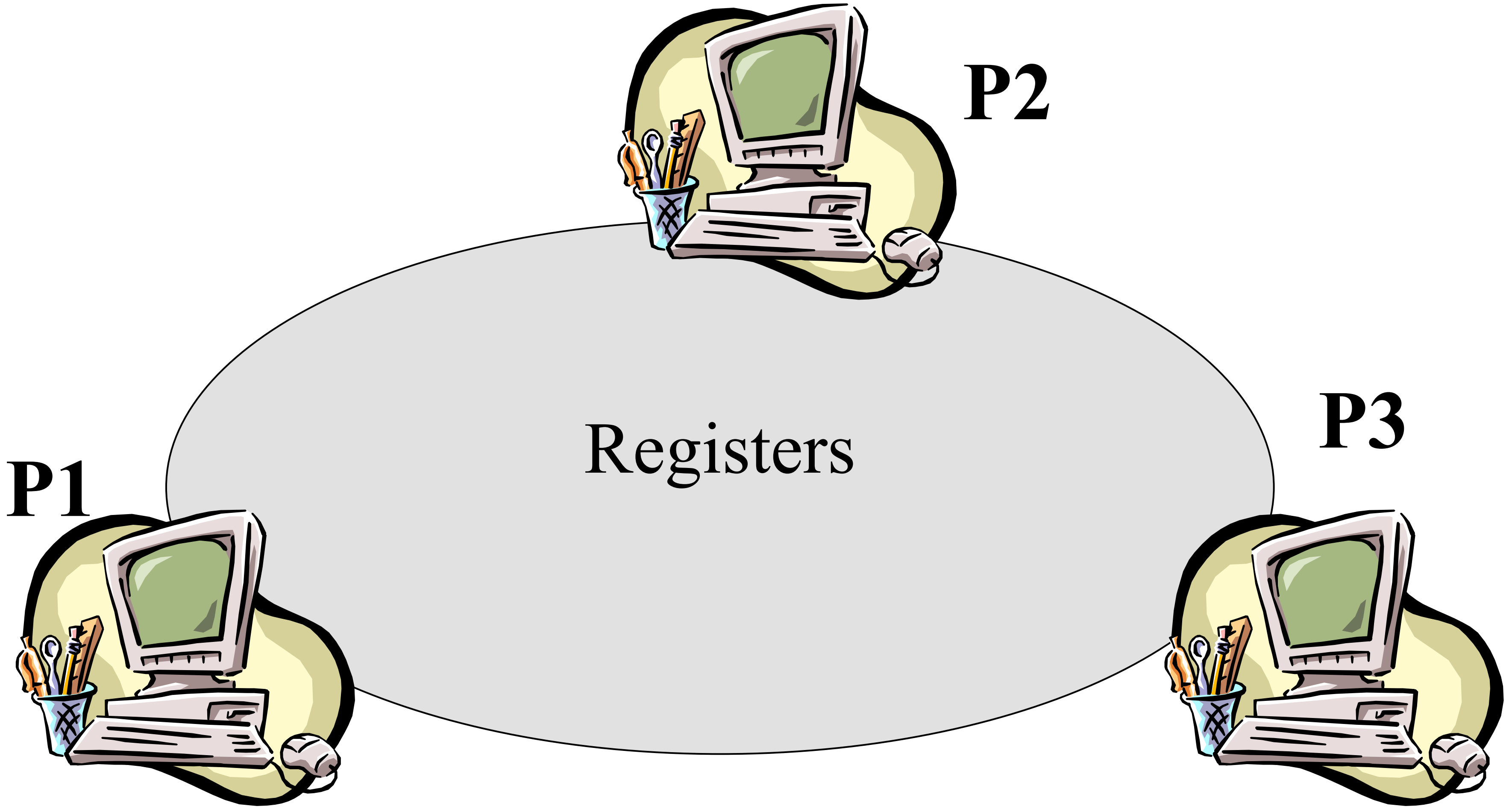
Overview of this lecture

- (1) ***Overview of a register algorithm***
- (2) ***A bogus algorithm***
- (3) ***A simplistic algorithm***
- (4) ***A simple fail-stop algorithm***
- (5) ***A tight asynchronous lower bound***
- (6) ***A fail-silent algorithm***

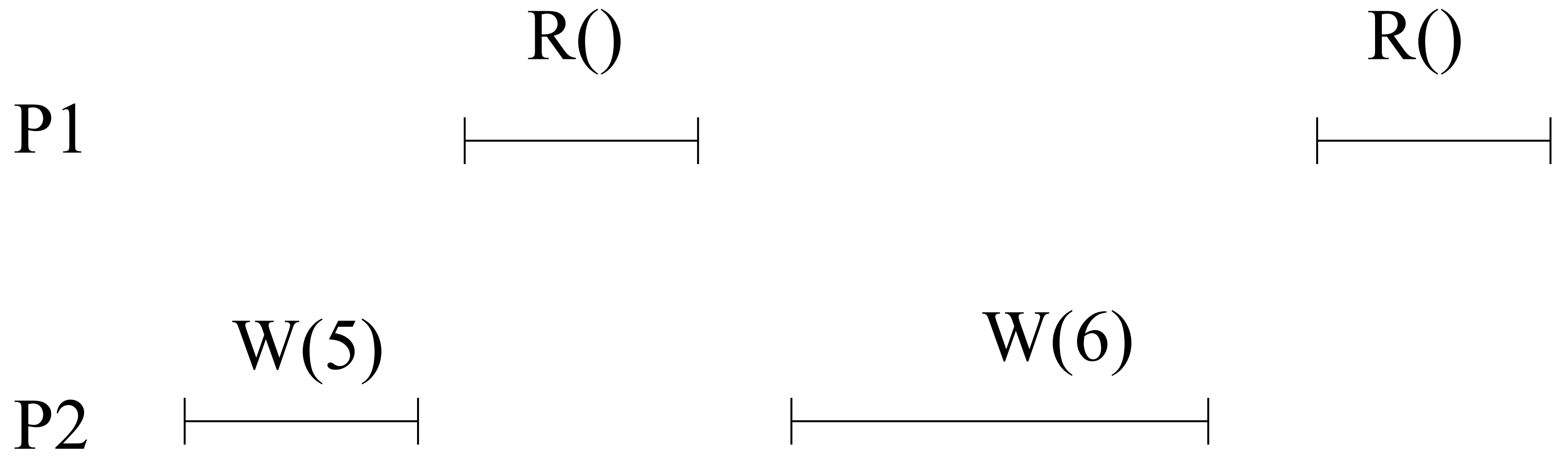
- Atomic register specification -

R. Guerraoui
Distributed Programming Laboratory
lpdwww.epfl.ch

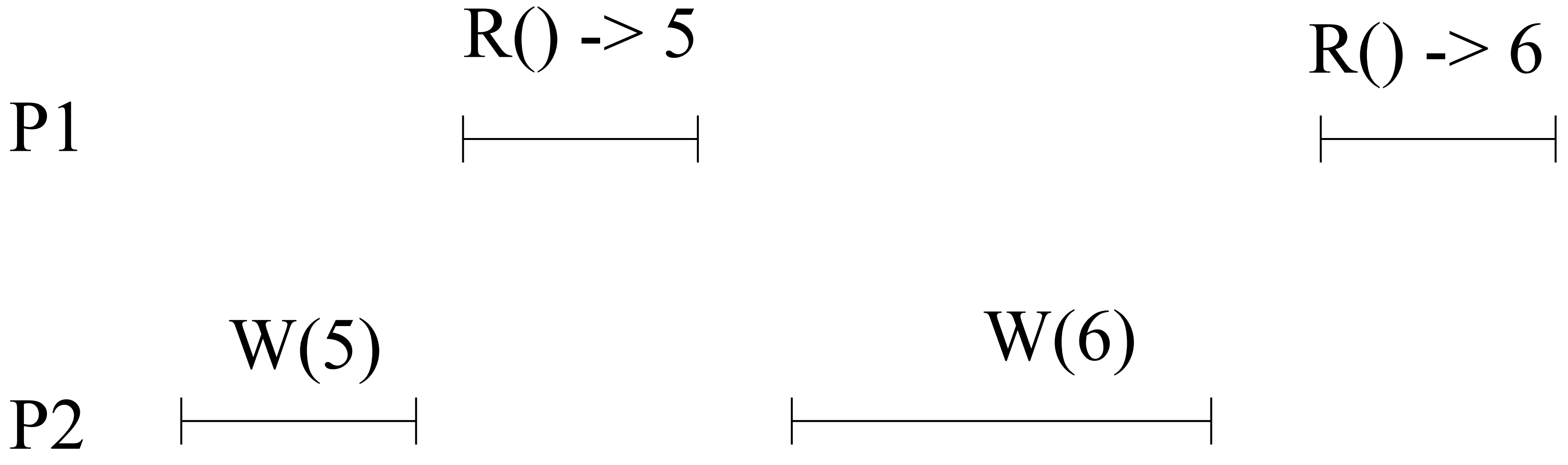
The application model



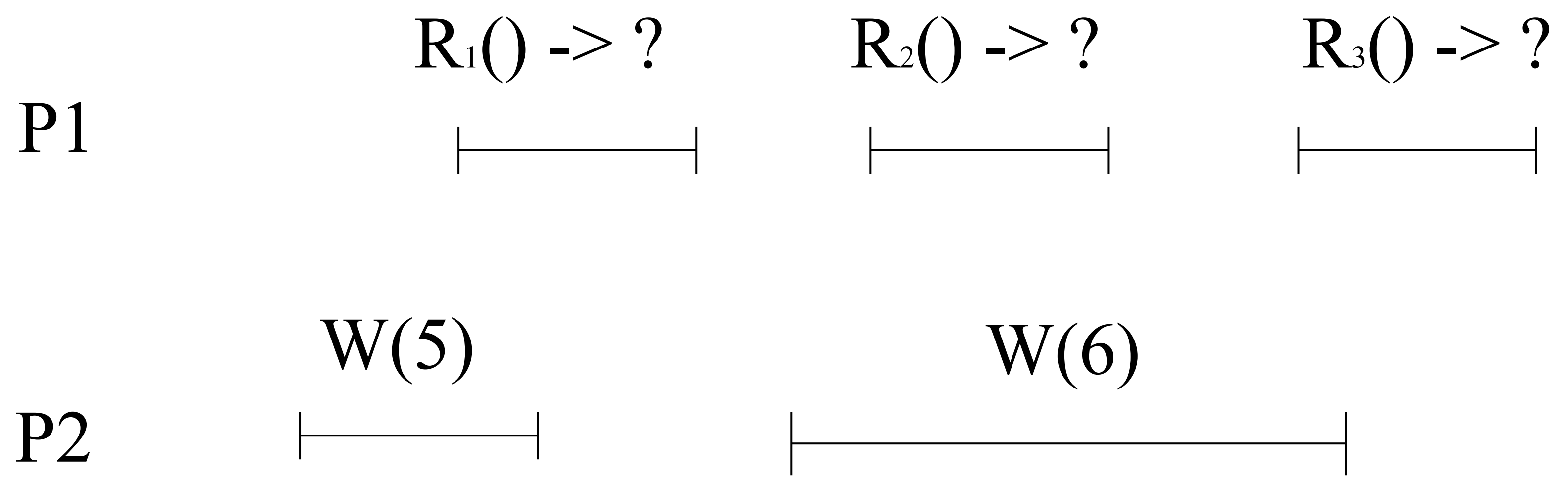
Sequential execution



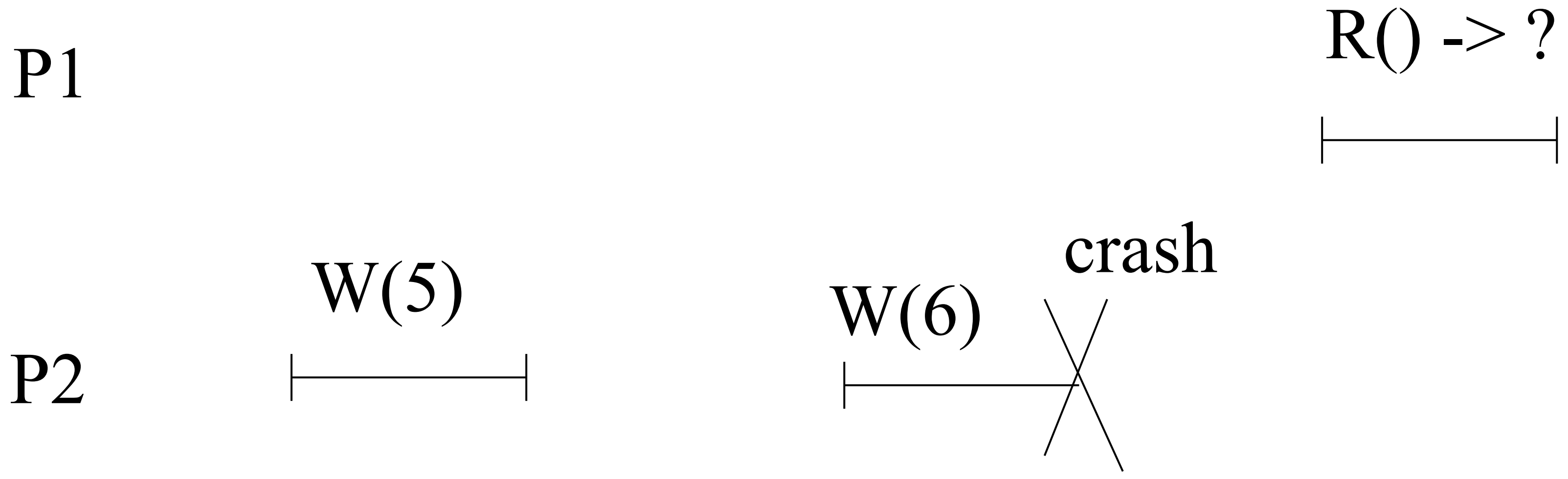
Sequential execution



Concurrent execution



Execution with failures



Safety

- ***An atomic register*** provides strong guarantees even when there is concurrency and failures

- The execution is equivalent to a sequential and failure-free execution (***linearization***)

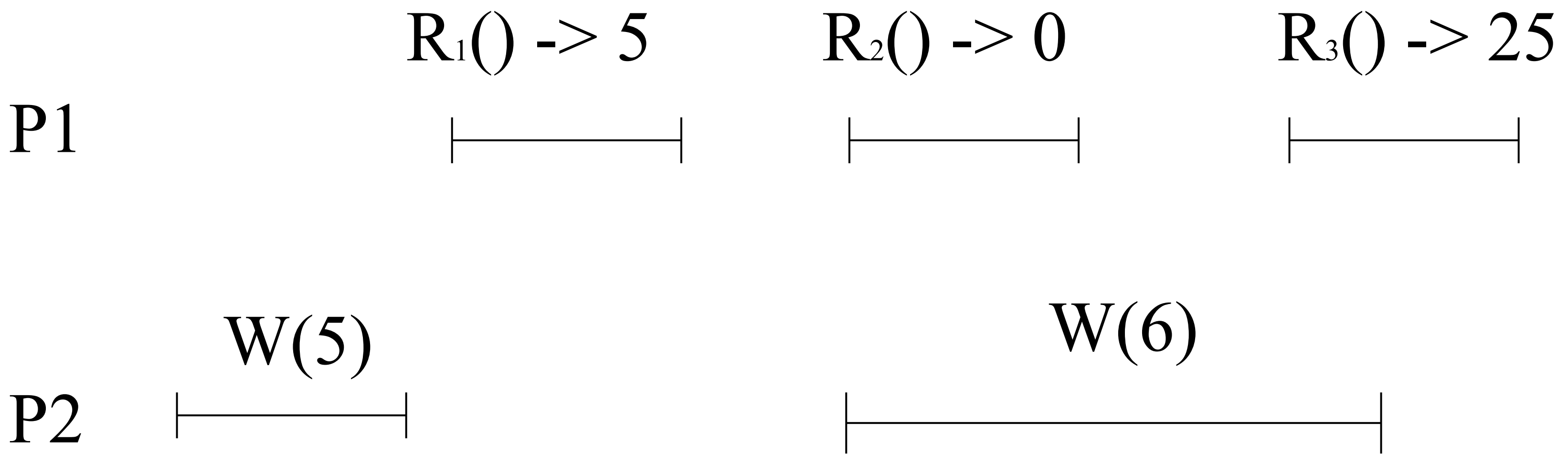
Atomic register

- Every failed (write) operation appears to be either complete or not to have been invoked at all

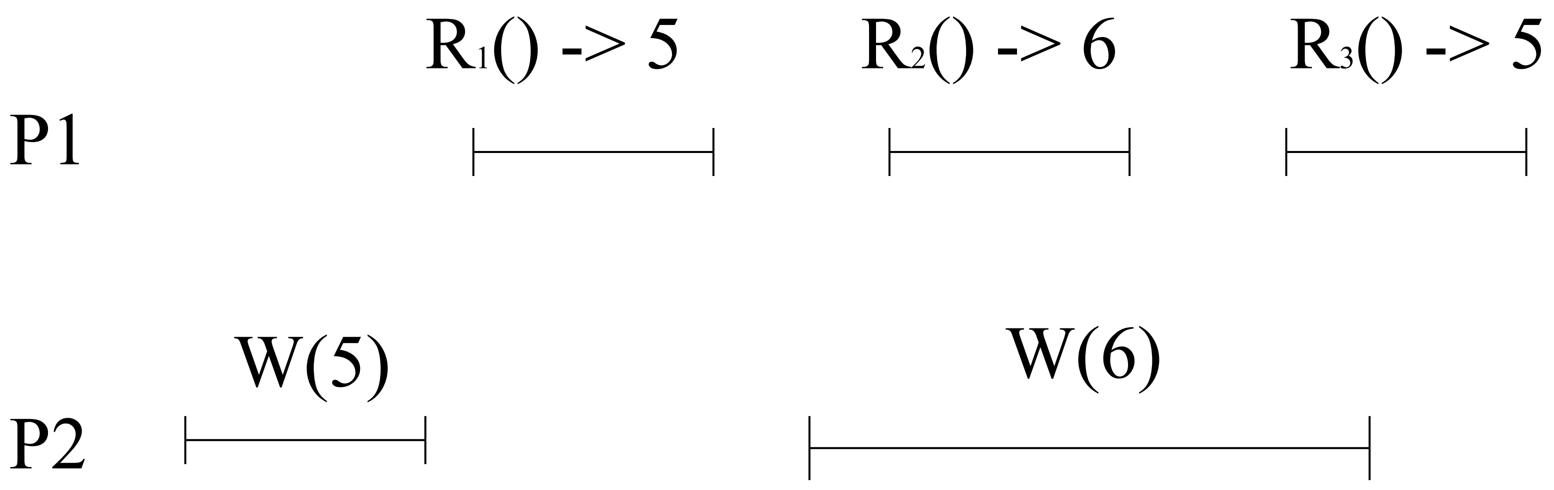
And

- Every complete operation appears to be executed at some instant between its invocation and reply time events

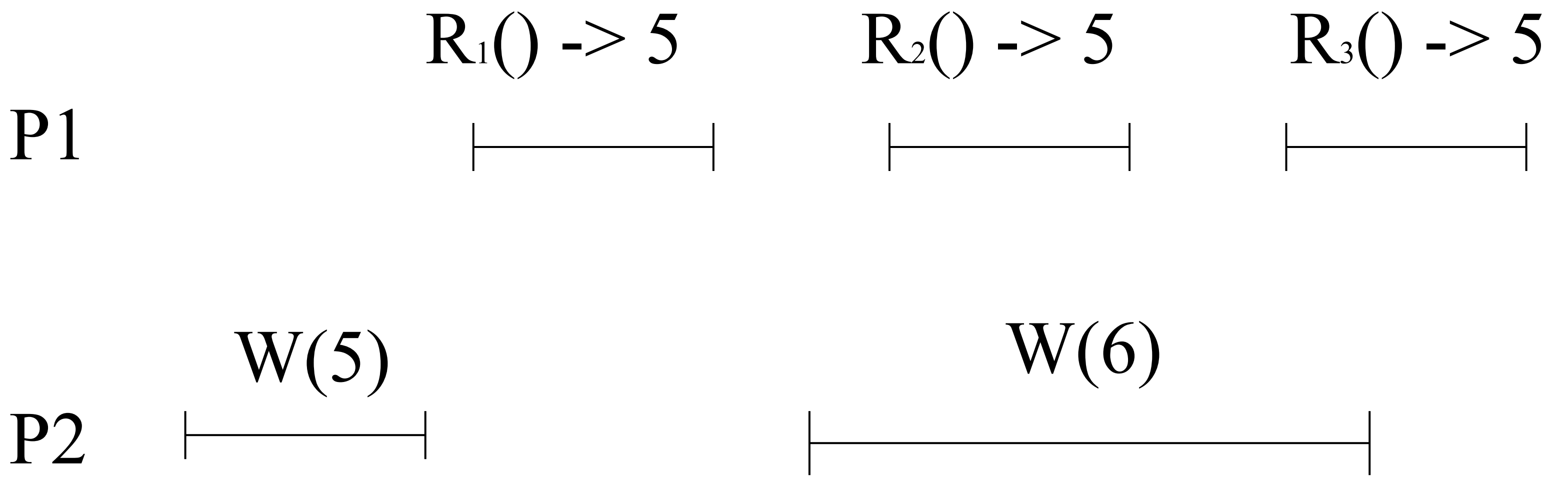
Execution 1



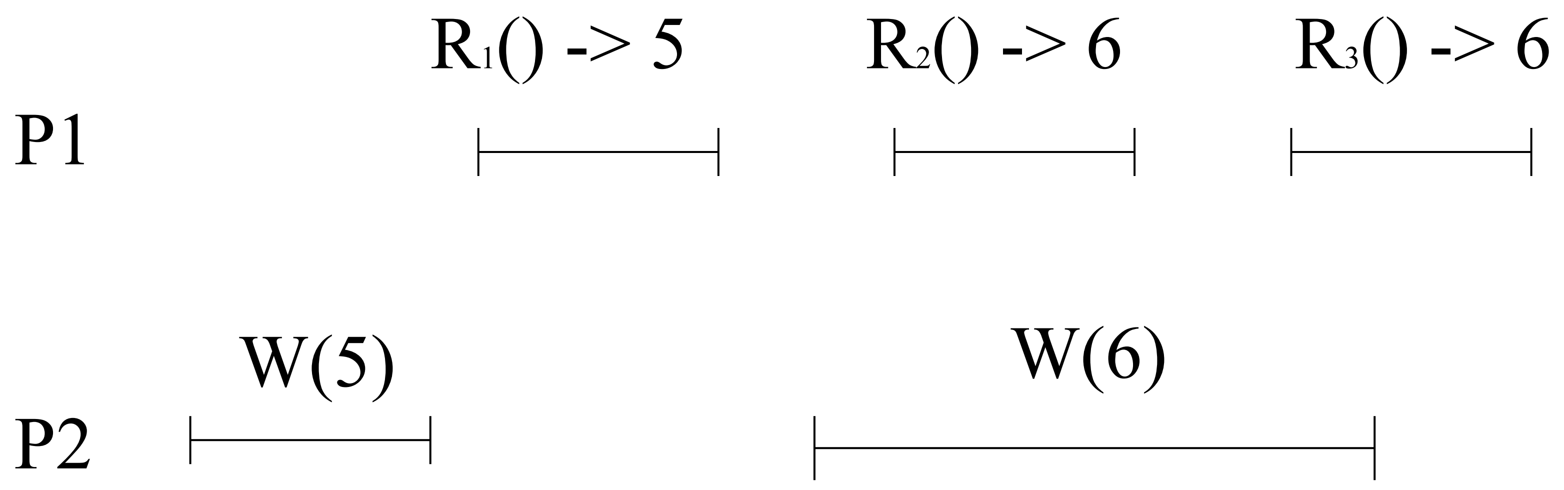
Execution 2



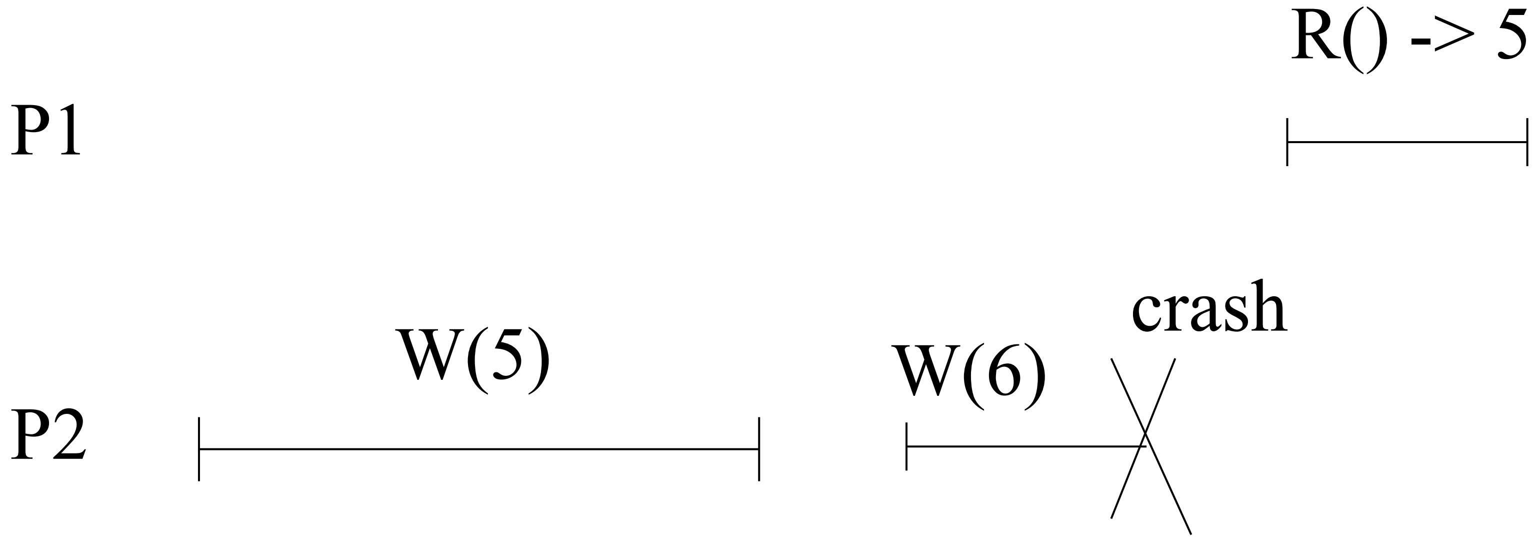
Execution 3



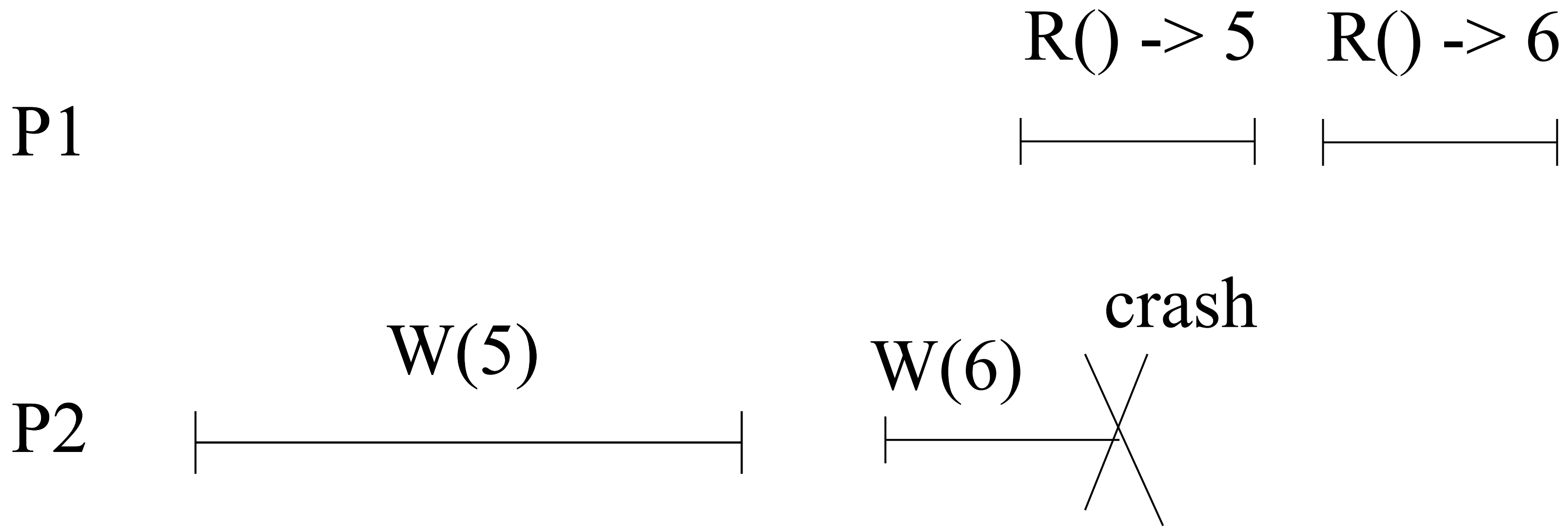
Execution 4



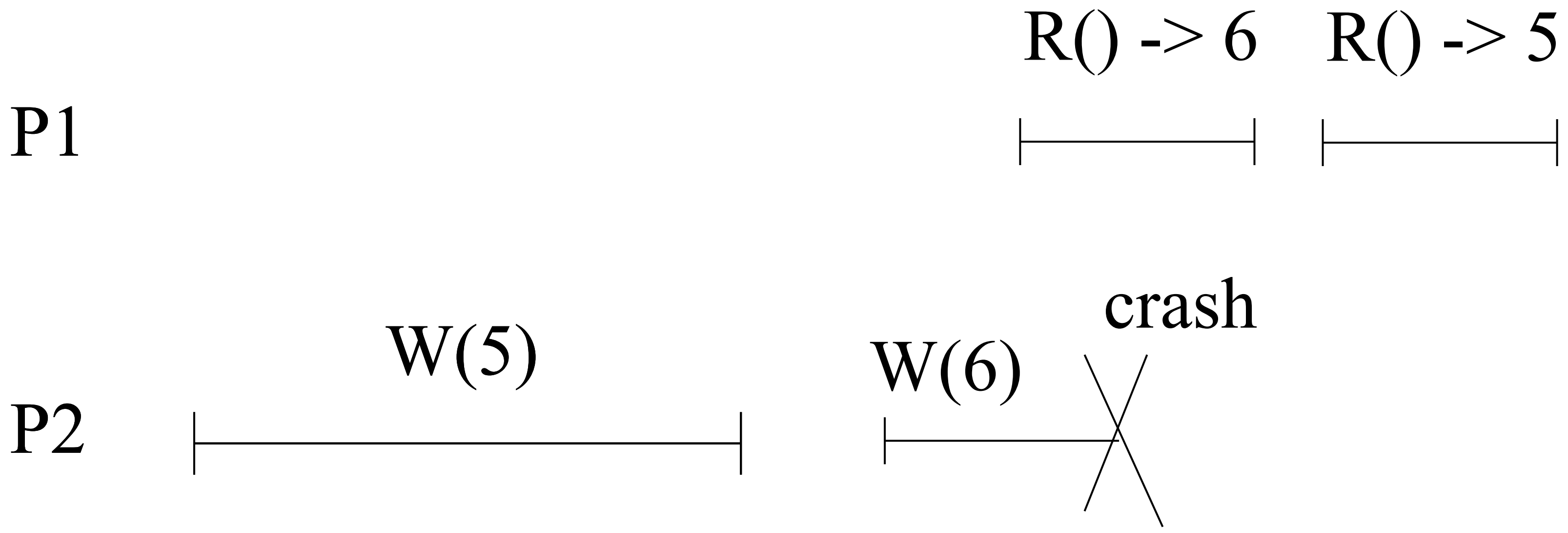
Execution 5



Execution 6



Execution 7



Correctness

- Execution 1: non-regular (safe)
- Executions 2 and 7: non-atomic (regular)
- Executions 3; 4, 5 and 6: atomic

Regular vs Atomic

- For a regular register to be atomic, two successive ***Read()*** must not overlap a ***Write()***

- The regular register might in this case allow the first ***Read()*** to obtain the new value and the second ***Read()*** to obtain the old value

Atomic register algorithms

R. Guerraoui

Distributed Programming Laboratory
lpdwww.epfl.ch

Overview of this lecture

- ***(1) From regular to atomic***
- ***(2) A 1-1 atomic fail-stop algorithm***
- ***(3) A 1-N atomic fail-stop algorithm***
- ***(4) A N-N atomic fail-stop algorithm***
- ***(5) From fail-stop to fail-silent***

Fail-stop algorithms

- We first assume a fail-stop model; more precisely:
  - any number of processes can fail by crashing (no recovery)
  - channels are reliable
  - failure detection is perfect

The simple algorithm

- Consider our fail-stop ***regular*** register algorithm
  - every process has a local copy of the register value
  - every process reads ***locally***
  - the writer writes ***globally,*** i.e., at all (non-crashed) processes

The simple algorithm

- Write(v) at pi
  - send [W,v] to all
  - for every pj, wait until either:
    - received [ack] or
    - suspected [pj]
  - Return ok

- At pi:

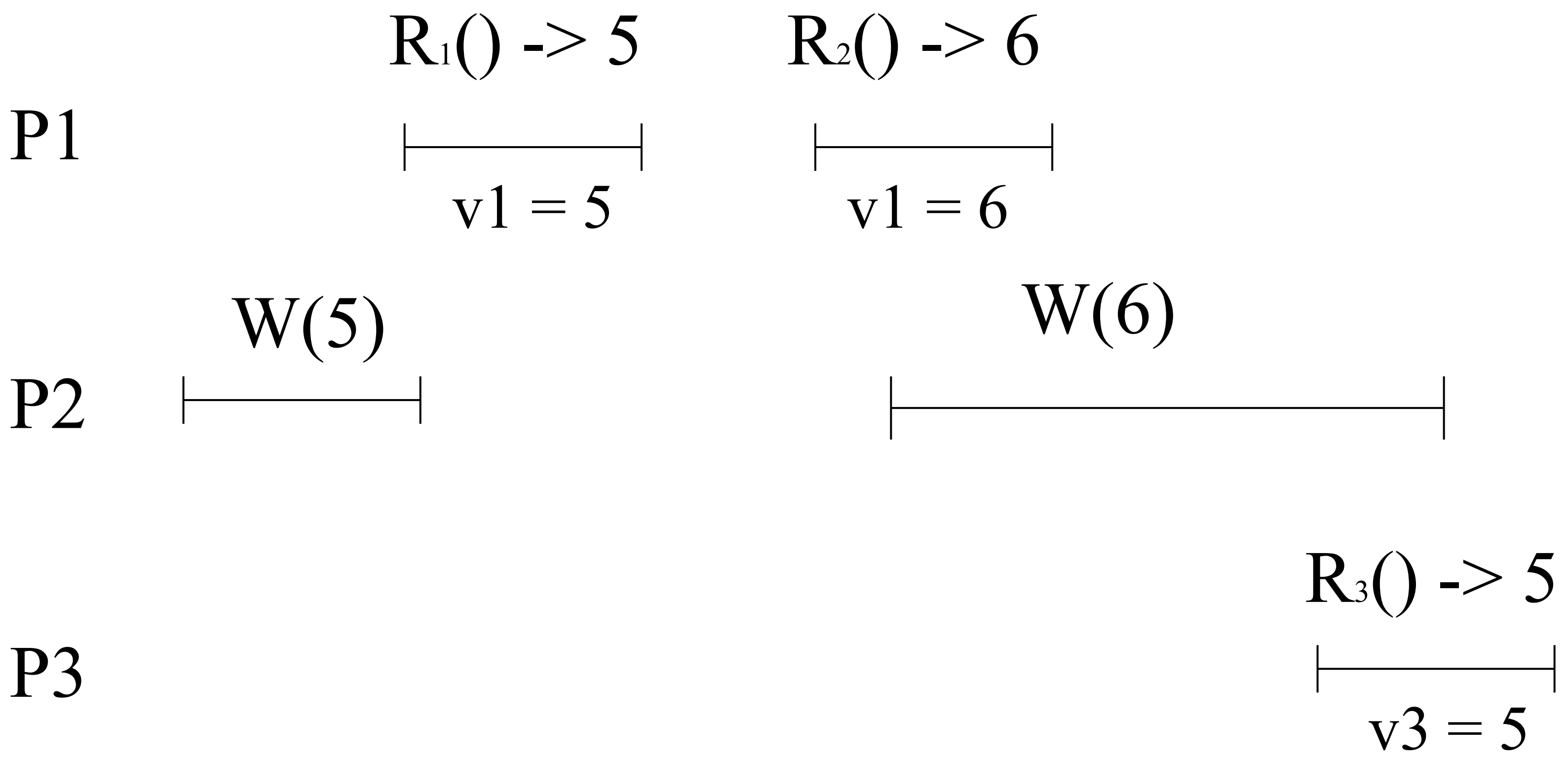
when receive [W,v] from pj

vi := v

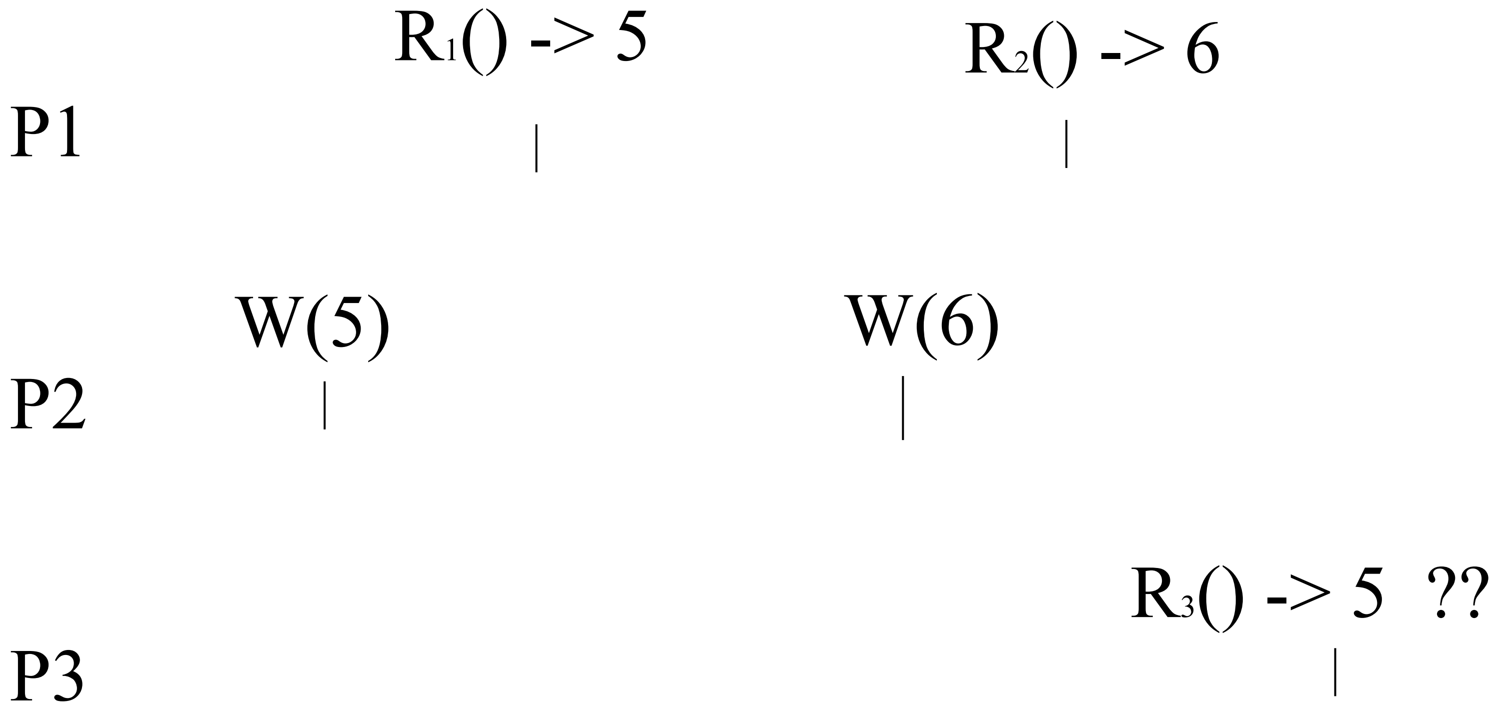
send [ack] to pj

- Read() at pi
  - Return vi

Atomicity?



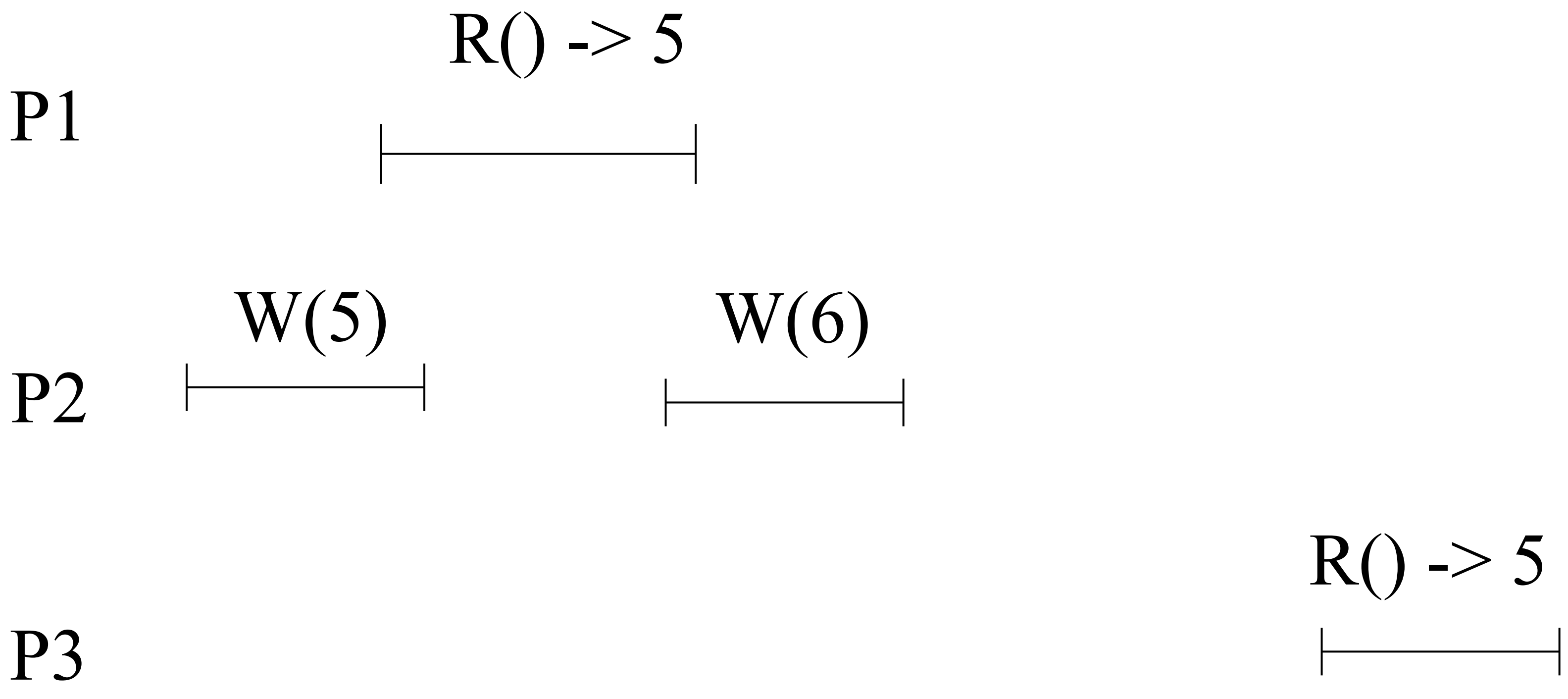
Linearization?



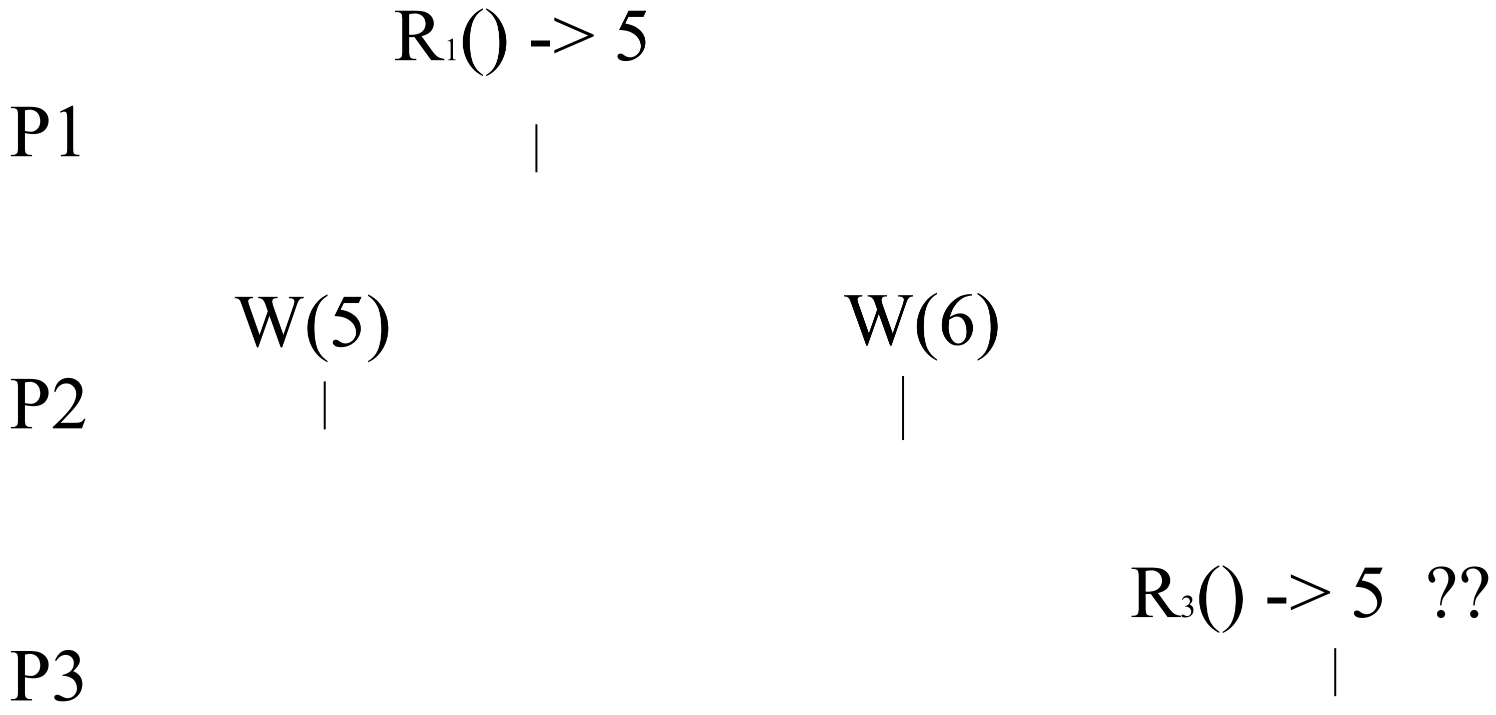
Fixing the pb: read-globally

- Read() at pi
  - send [W,vi] to all
  - for every pj, wait until either:
    - receive [ack] or
    - suspect [pj]
  - Return vi

Still a problem



Linearization?



Overview of this lecture

- ***(1) From regular to atomic***
- ***(2) A 1-1 atomic fail-stop algorithm***
- ***(3) A 1-N atomic fail-stop algorithm***
- ***(4) A N-N atomic fail-stop algorithm***
- ***(5) From fail-stop to fail-silent***

A fail-stop 1-1 atomic algorithm

- Write(v) at p1
  - send [W,v] to p2
  - Wait until either:
    - receive [ack] from p2 or
    - suspect [p2]
  - Return ok

- At p2:

  when receive [W,v] from p1

  v2 := v

  send [ack] to p2

- Read() at p2
  - Return v2

A fail-stop 1-N algorithm

- every process maintains a local value of the register as well as a sequence number
- the writer, p1, maintains, in addition a timestamp ts1
- any process can read in the register

A fail-stop 1-N algorithm

- Write(v) at p1
  - ts1++
  - send [W,ts1,v] to all
  - for every pi, wait until either:
    - receive [ack] or
    - suspect [pi]
  - Return ok

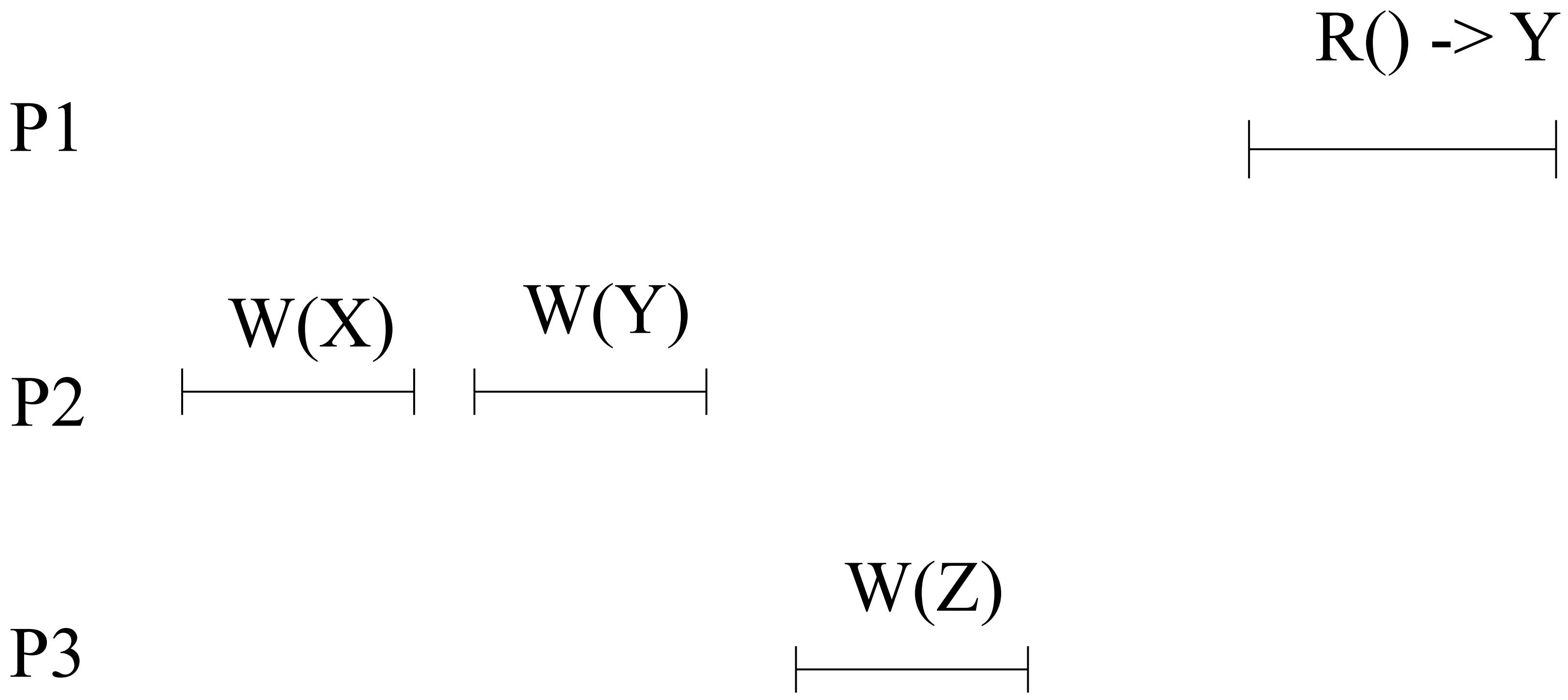
- Read() at pi
  - send [W,sni,vi] to all
  - for every pj, wait until either:
    - receive [ack] or
    - suspect [pj]
  - Return vi

A 1-N algorithm (cont’d)

- At pi
  - When pi receive [W,ts,v] from pj

```
        if ts > sni then
              vi := v
              sni := ts
        send [ack] to pj
```

Why not N-N?



The Write() algorithm

- Write(v) at pi
  - ✓ send [W] to all
  - ✓ for every pj wait until
    - **receive [W,snj] or**
    - **suspect pj**
  - ✓ (sn,id) := (highest snj + 1,i)
  - ✓ send [W,(sn,id),v] to all
  - ✓ for every pj wait until
    - **receive [W,(sn,id),ack] or**
    - **suspect pj**
  - ✓ Return ok

- At pi

  T1:
  - ✓ when receive [W] from pj
    - send [W,sn] to pj

  T2:
  - ✓ when receive [W,(snj,idj),v] from pj
  - ✓ If (snj,idj) > (sn,id) then
    - vi := v
    - (sn,id) := (snj,idj)
  - ✓ send [W,(snj,idj),ack] to pj

The Read() algorithm

- Read() at pi
  - ✓ send [R] to all
  - ✓ for every pj wait until
    - **receive [R,(snj,idj),vj] or**
    - **suspect pj**
  - ✓ v = vj with the highest (snj,idj)
  - ✓ (sn,id) = highest (snj,idj)
  - ✓ send [W,(sn,id),v] to all
  - ✓ for every pj wait until
    - **receive [W,(sn,id),ack] or**
    - **suspect pj**
  - ✓ Return v

- At pi

  T1:
  - ✓ when receive [R] from pj
    - send [R,(sn,id),vi] to pj

  T2:
  - ✓ when receive [W,(snj,idj),v] from pj
  - ✓ If (snj,idj) > (sn,id) then
    - vi := v
    - (sn,id) := (snj,idj)
  - ✓ send [W,(snj,idj),ack] to pj

Overview of this lecture

- ***(1) From regular to atomic***
- ***(2) A 1-1 atomic fail-stop algorithm***
- ***(3) A 1-N atomic fail-stop algorithm***
- ***(4) A N-N atomic fail-stop algorithm***
- ***(5) From fail-stop to fail-silent***

From fail-stop to fail-silent

- We assume a majority of correct processes
- In the 1-N algorithm, the writer writes in a majority using a timestamp determined locally and the reader selects a value from a majority and then imposes this value on a majority
- In the N-N algorithm, the writers determines first the timestamp using a majority